**《媒体数据管理》上机实验报告**

注：源代码中Corel数据下载地址：

<http://www.ci.gxnu.edu.cn/cbir/Corel.zip>

姓名：张笑天

学号：17030140014

目录

[第1次实验 算术编码 2](#_Toc43570060)

[算法描述： 2](#_Toc43570061)

[核心源程序： 2](#_Toc43570062)

[测试数据： 6](#_Toc43570063)

[第2次实验 K-L变换和矢量量化 6](#_Toc43570064)

[算法描述： 6](#_Toc43570065)

[核心源程序： 6](#_Toc43570066)

[测试数据： 7](#_Toc43570067)

[测试数据： 10](#_Toc43570068)

[第3次实验 局部敏感哈希 11](#_Toc43570069)

[算法描述： 11](#_Toc43570070)

[核心源程序： 11](#_Toc43570071)

[测试数据： 13](#_Toc43570072)

[第4次实验 SIFT 13](#_Toc43570073)

[核心源程序： 15](#_Toc43570074)

[第5次实验 KDTree 19](#_Toc43570075)

[算法描述： 19](#_Toc43570076)

[核心源程序： 20](#_Toc43570077)

[测试数据： 22](#_Toc43570078)

[个人感想 23](#_Toc43570079)

# 第1次实验 算术编码

## 算法描述：

参考：

<https://www.drdobbs.com/cpp/data-compression-with-arithmetic-encodin/240169251?pgno=1>

与传统方法不同，使用整数来构造编码。

本质上利用比特位全为1的整数，例如32位无符号整数，来模拟无限长1的数字，从而表示浮点数1，即利用0XFFFFFFFF模拟0.FFFFFFFF…；同理，利用比特全为0的整数表示浮点数0 。

接下来编码过程和浮点数类似，通过编码范围构造新的整数和整数，但是为了保证可以编码解码无限长的数字，每次构造之后，所有参与编码数字左移一位。同时，为了防止在编码范围以01和10开头时，高低范围聚合，会进行一些特殊处理。

解码过程位构造同样位数的整数，以类似流的形式读入编码，构造当前表示的整数和整数范围，得到表示的字母，并且同时模拟编码过程；之后同样参与解码数字左移一位，得到新的整数和整数范围。

实现使用C++，编译器为MSVC，可能采用某些modern C++语法。

## 核心源程序：

编码函数：

std::string encode(std::string s, std::map<char, std::pair<int, int>> table, int denominator)

{

s\_c = s;

CODEVALUE high{ CODEMAX };

CODEVALUE low{ CODEMIN };

int pending\_bits{ 0 };

std::string ret;

for (char c : s)

{

// Similiar to floating coding.

CODEVALUE range{ high - low + 1};

int lower{ table[c].first };

int upper{ table[c].second };

// Overflow avoidance.

high = low + range \* upper / denominator - 1;

low = low + range \* lower / denominator;

for (; ; )

{

// Looking for situations in which the values have the same most-significant bit.

// MSB is 0.

if (high < HALFMAX)

{

ret += "0";

for (int i{ 0 }; i < pending\_bits; i++)

ret += "1";

pending\_bits = 0;

}

// MSB is 1.

else if (low >= HALFMAX)

{

ret += "1";

for (int i{ 0 }; i < pending\_bits; i++)

ret += "0";

pending\_bits = 0;

}

// The situation that high and low are converging,

// when low starts with 10 and high starts with 01.

else if (low >= ONEFORTH && high < THIRDFORTH)

{

// Discard the second most-significant bit,

// shifting the remaining bits left, while retaining the MSB,

// doing that means also incrementing the pending\_bits counter to acknowledge

// that we need to deal with it when convergence finally happens.

pending\_bits++;

low -= ONEFORTH;

high -= ONEFORTH;

}

else

{

break;

}

high <<= 1;

high++;

low <<= 1;

high &= CODEMAX;

low &= CODEMAX;

}

// The last bit.

pending\_bits++;

if (low < ONEFORTH)

{

ret += "0";

for (int i{ 0 }; i < pending\_bits; i++)

ret += "1";

pending\_bits = 0;

}

else

{

ret += "1";

for (int i{ 0 }; i < pending\_bits; i++)

ret += "0";

pending\_bits = 0;

}

}

return ret;

}

解码函数：

std::string decode(std::string s, std::map<char, std::pair<int, int>> table, int denominator)

{

CODEVALUE high{ CODEMAX };

CODEVALUE low{ CODEMIN };

CODEVALUE value{ CODEMIN };

std::string ret{ s\_c };

// Value has the bits from the encoded message shifted into.

for (unsigned int i{ 0 }; i < sizeof(unsigned int) \* 8; i++)

{

value <<= 1;

if (s.length() != 0)

{

value += (s.at(0) == '1' ? 1 : 0);

s = s.substr(1);

}

}

for ( ; ; )

{

CODEVALUE range{ high - low + 1 };

CODEVALUE count = ((value - low + 1) \* denominator - 1) / range;

char c;

// Find the letter that in the range of count.

std::map<char, std::pair<int, int>>::iterator iter;

iter = table.begin();

while (iter != table.end())

{

if (count >= static\_cast<unsigned int>(iter->second.first) && count < static\_cast<unsigned int>(iter->second.second))

{

c = iter->first;

break;

}

iter++;

}

// ret += c;

// Repeat encoder.

int lower{ table[c].first };

int upper{ table[c].second };

high = low + range \* upper / denominator - 1;

low = low + range \* lower / denominator;

for (; ; )

{

if (high < HALFMAX)

{

;

// Do nothing, bit is a zero

}

else if (low >= HALFMAX)

{

value -= HALFMAX;

low -= HALFMAX;

high -= HALFMAX;

}

else if (low >= ONEFORTH && high < THIRDFORTH)

{

value -= ONEFORTH;

low -= ONEFORTH;

high -= ONEFORTH;

}

else

{

break;

}

low <<= 1;

high <<= 1;

high++;

value <<= 1;

if (s.length() != 0)

{

value += (s.at(0) == '1' ? 1 : 0);

s = s.substr(1);

}

}

break;

}

return ret;

}

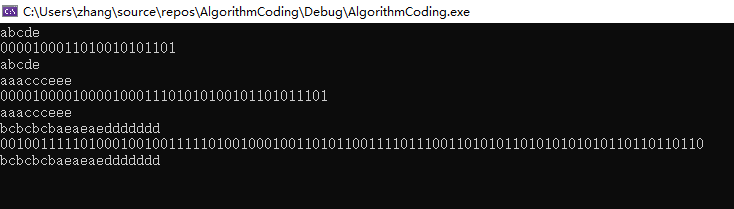
## 测试数据：

格式：

字符串：只录入了五个字符a b c d e

编码结果

解码结果



# 第2次实验 K-L变换和矢量量化

K-L变换

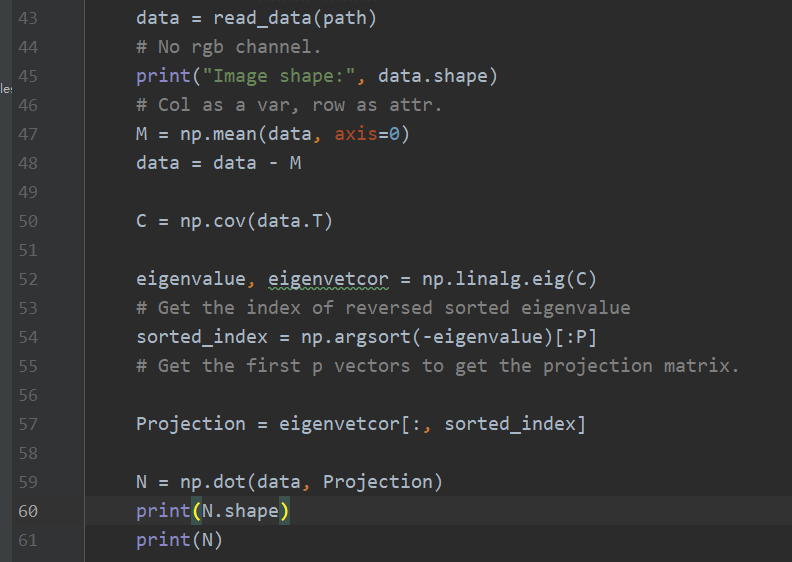
## 算法描述：

采用PCA算法。

对于读入矩阵，进行减去每个元素均值，求方差矩阵，求特征向量和特征值，对特征向量按照特征值排序，取特征值最大的前n个特征向量组成投影矩阵，将原来数据与投影矩阵相乘得到n维新的数据。

代码实现采用Python3

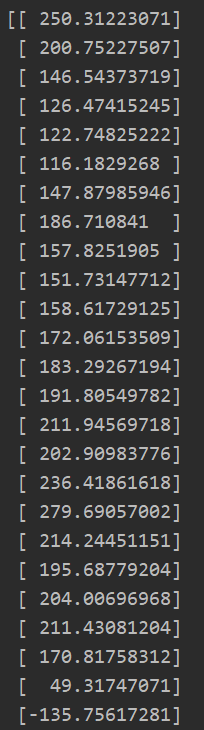
## 核心源程序：



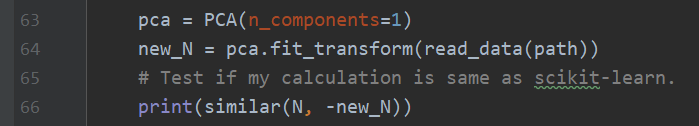
## 测试数据：

输入：Corel100类库/0\_1.jpg

输出：



最后验证是否与sklearn的PCA算法结果相同：

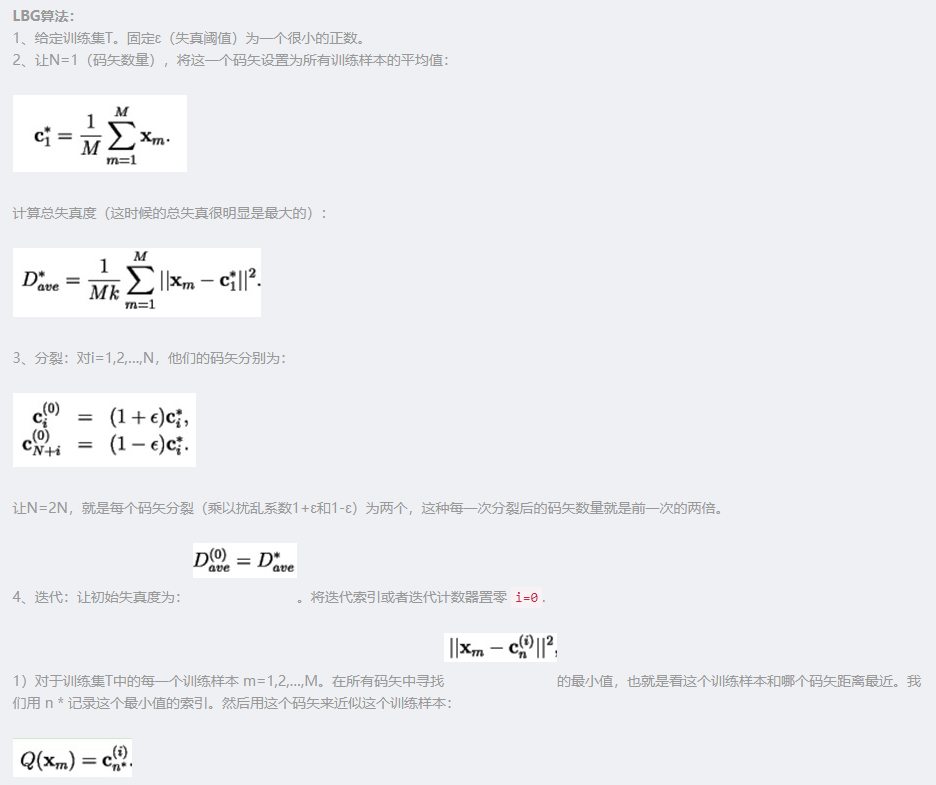


结果相同，证明算法正确。



矢量量化：

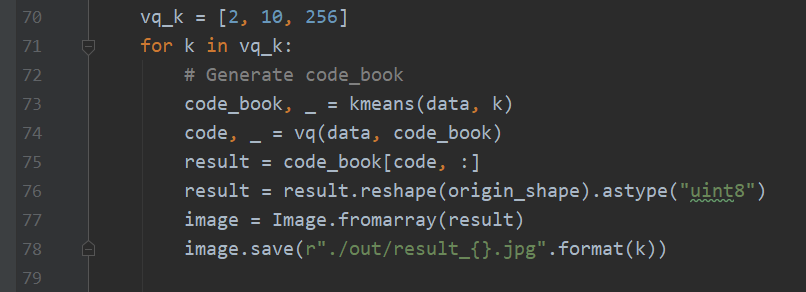
算法描述：





核心源程序：

使用sklearn包的vq算法



## 测试数据：

输入：Corel100类库/0\_1.jpg

输出：

原图：



k=2



k=10



k=256



# 第3次实验 局部敏感哈希

## 算法描述：

理论依据参考论文与：

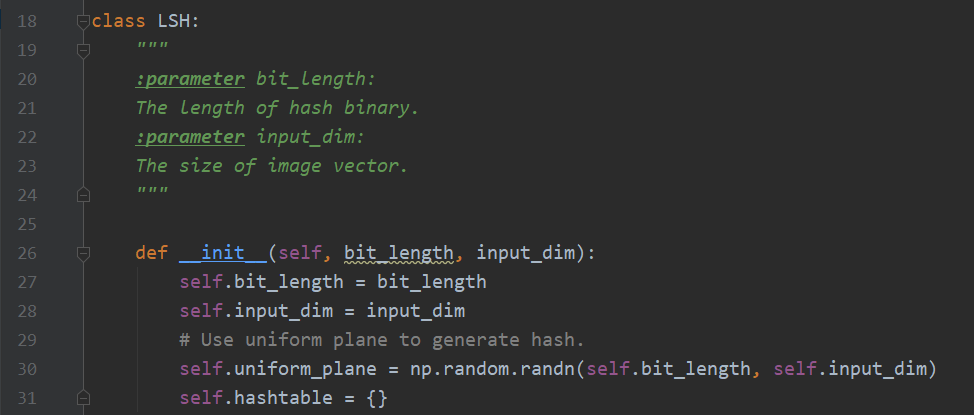
<https://towardsdatascience.com/understanding-locality-sensitive-hashing-49f6d1f6134>

本实现中，利用二维高斯矩阵与数据相乘构建哈希值，利用Python3字典数据结构构造哈希表，利用顺序数据结构来存储哈希冲突的数据。利用汉明编码判断两个哈希值距离。

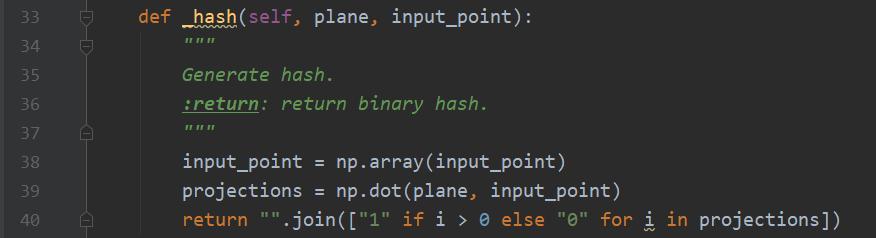
## 核心源程序：

构造LSH类：

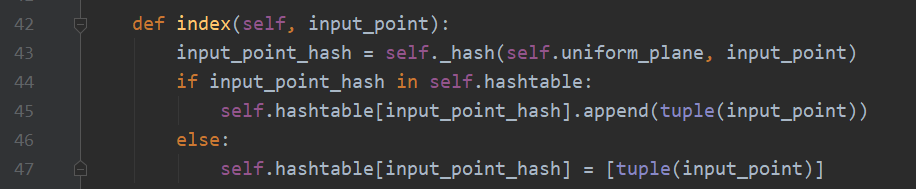
Uniform\_plane矩阵采用二维高斯分布。



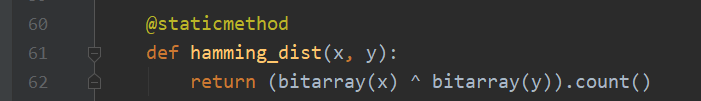
构造哈希值，其中plane矩阵为bit\_length \* dim维，bit\_length为哈希值长度，dim为数据长度。



将数据存入表中

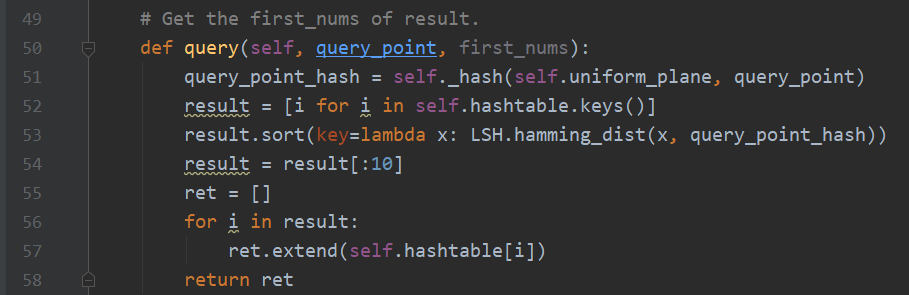


汉明距离：



查询数据：

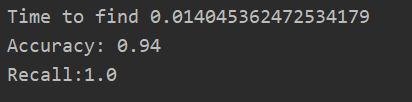
查找前十距离的哈希值。



## 测试数据：

随机产生100个数据，其中正例和反例概率均为0.5。

从上到下分别为平均查询时间、准确率、召回率。



# 第4次实验 SIFT

算法描述：

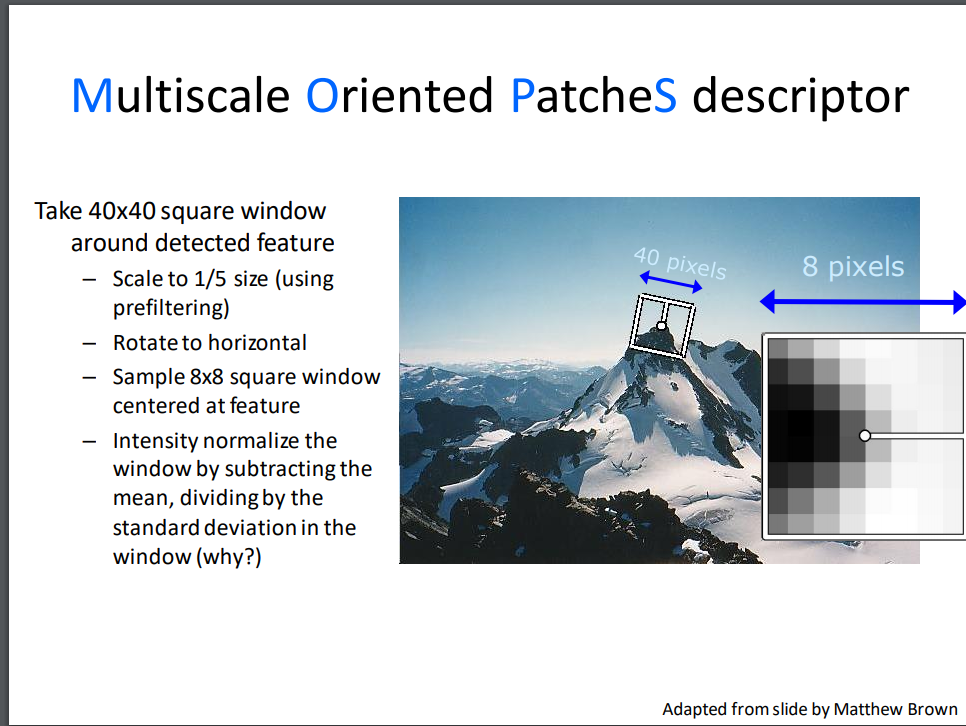
本实验中部分算法已经在计算机视觉课程中学习过。

自己实现了Harris角点检测算法，MOPS特征描述算法，RatioSSD和SSD匹配算法。

这三类算法均参考康奈尔大学课程PPT，分别为class 3 4 5 6 7 。

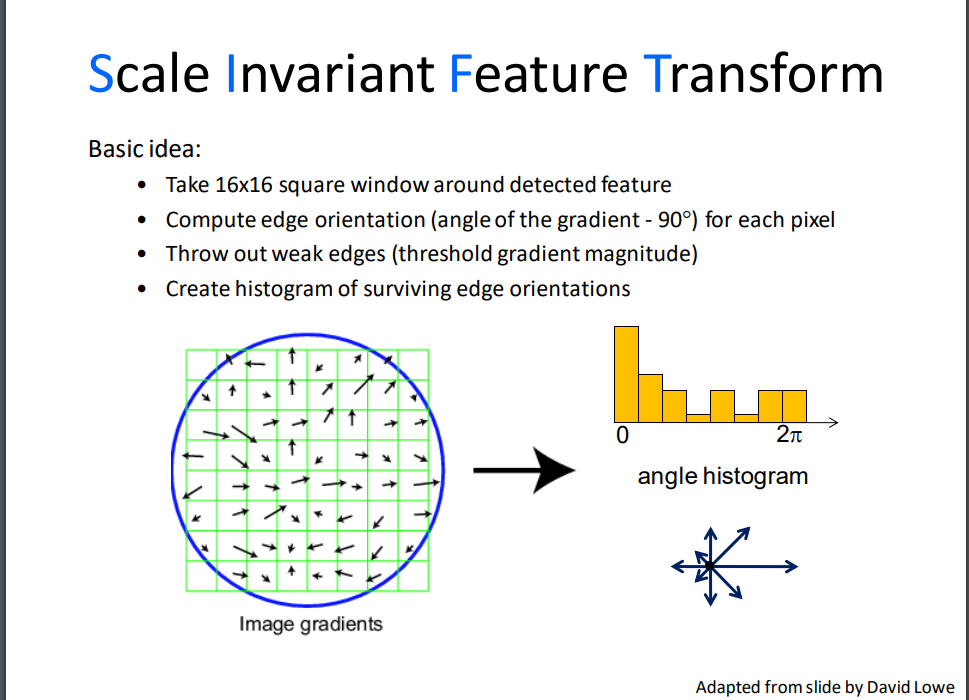
<http://www.cs.cornell.edu/courses/cs5670/2019sp/lectures/lectures.html>

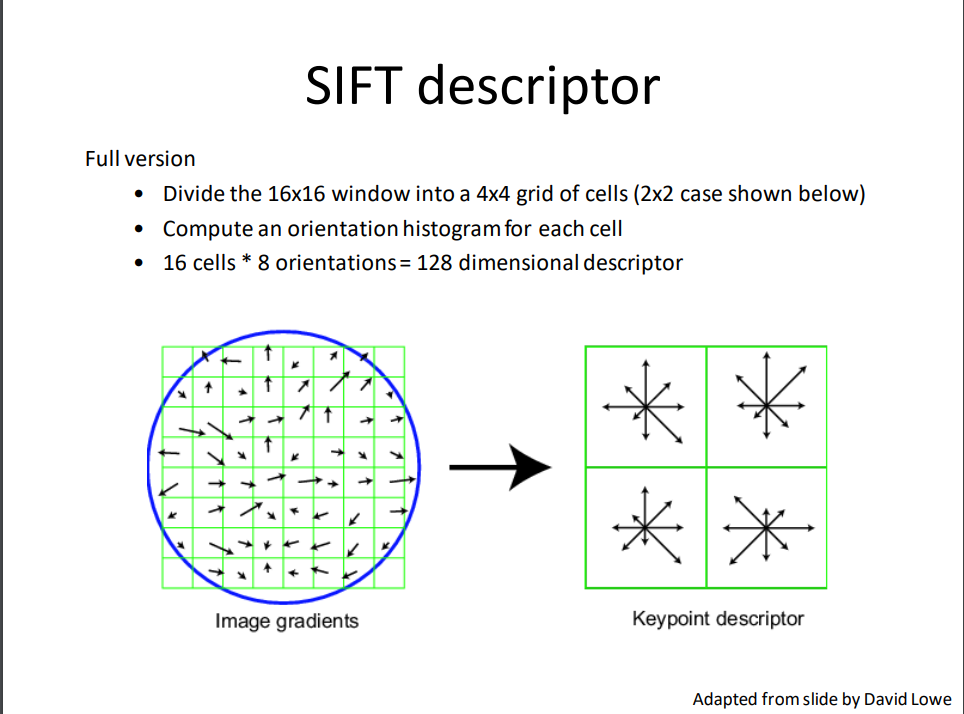
MOPS原理：



SIFT调用opencv-python库，匹配算法采用蛮力匹配。

SIFT原理参见论文和下图：





示例程序采用了Harris+MOPS+Ratio SSD，采用匹配点最多作为最相近。

也采用了SIFT+BFMatch，采用匹配总距离最小作为最相近，进行比较。

同时，也提供了其他算法，位于feature.py中，以下是本程序中可以使用算法：

特征点检测：

Harris，自己实现

ORB：调用opencv

特征描述：

MOPS：自己实现

SIFT：调用opencv

Simple：自己实现

特征匹配：

SSD：自己实现

RatioSSD：自己实现

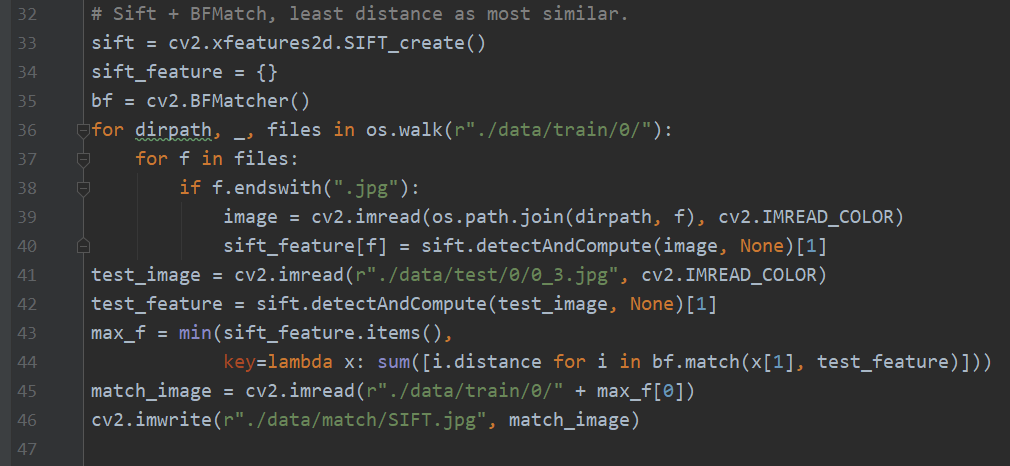
ORB：调用opencv

BFMatch：调用opencv

FlannMatch：调用opencv

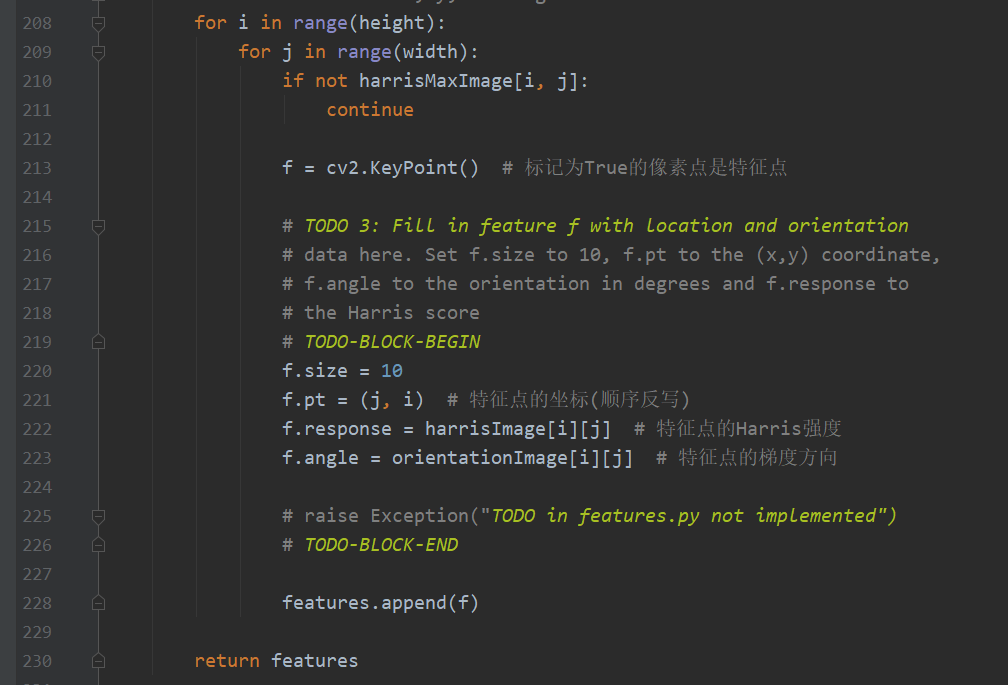
## 核心源程序：

SIFT采用opencv库。

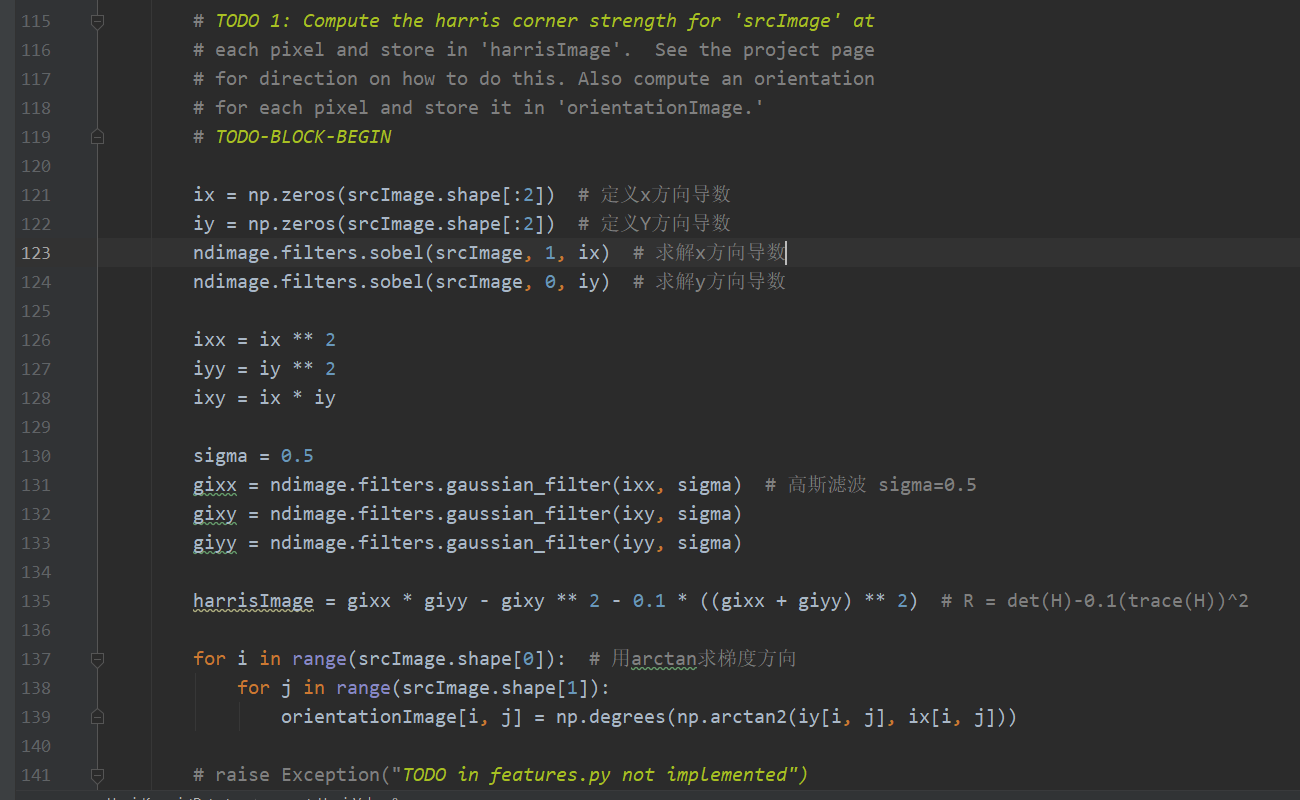


Harris：

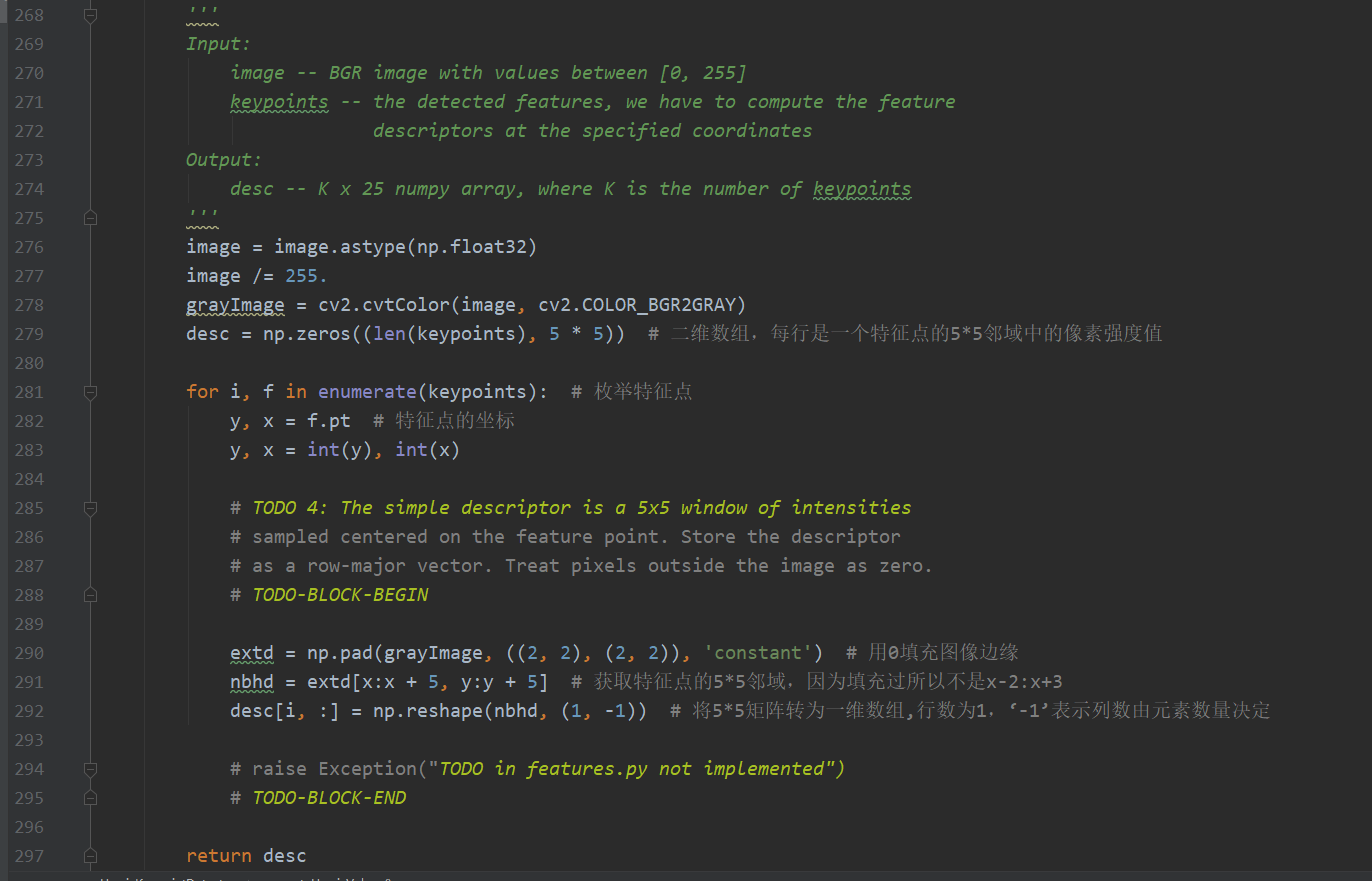
主函数



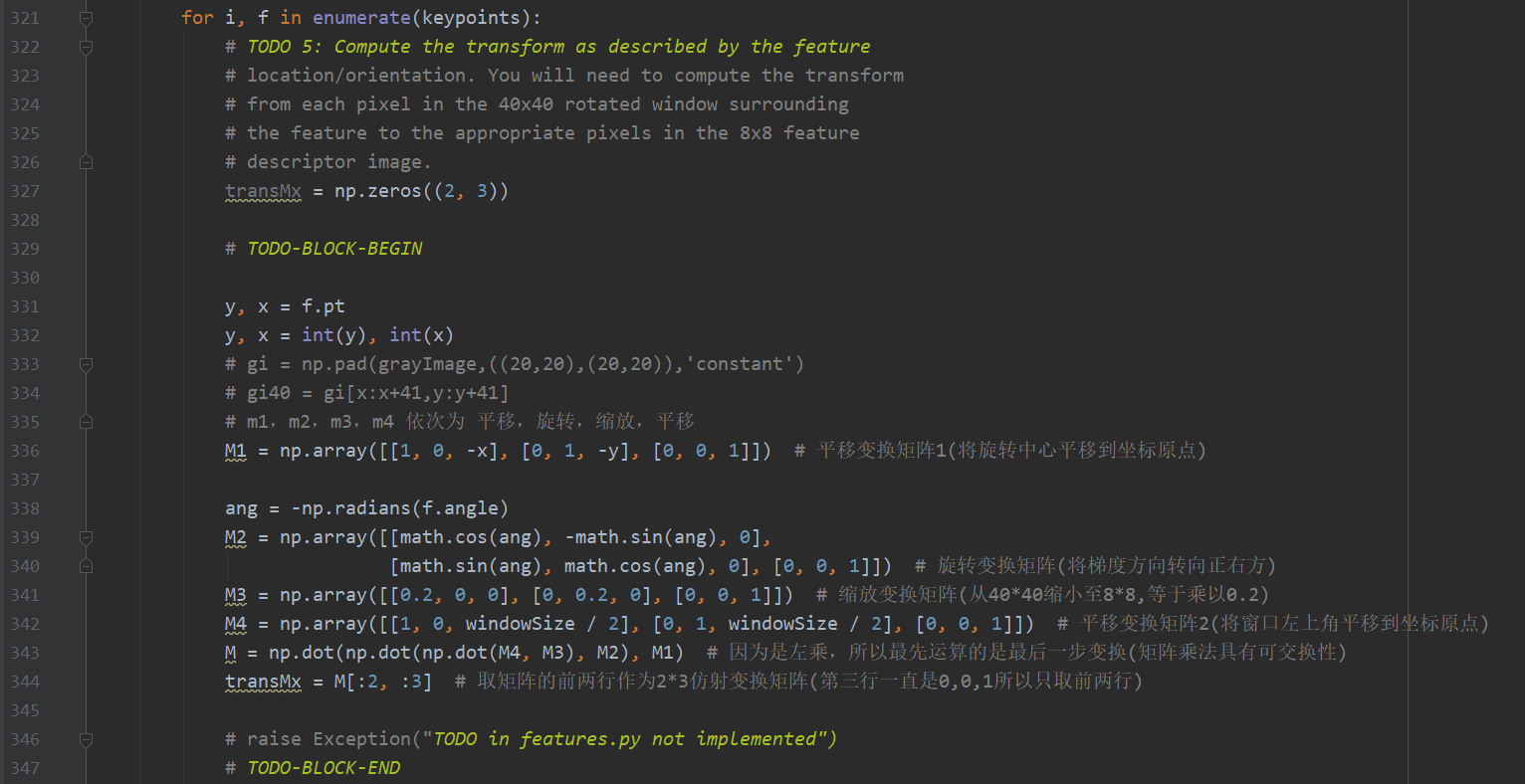
求解梯度

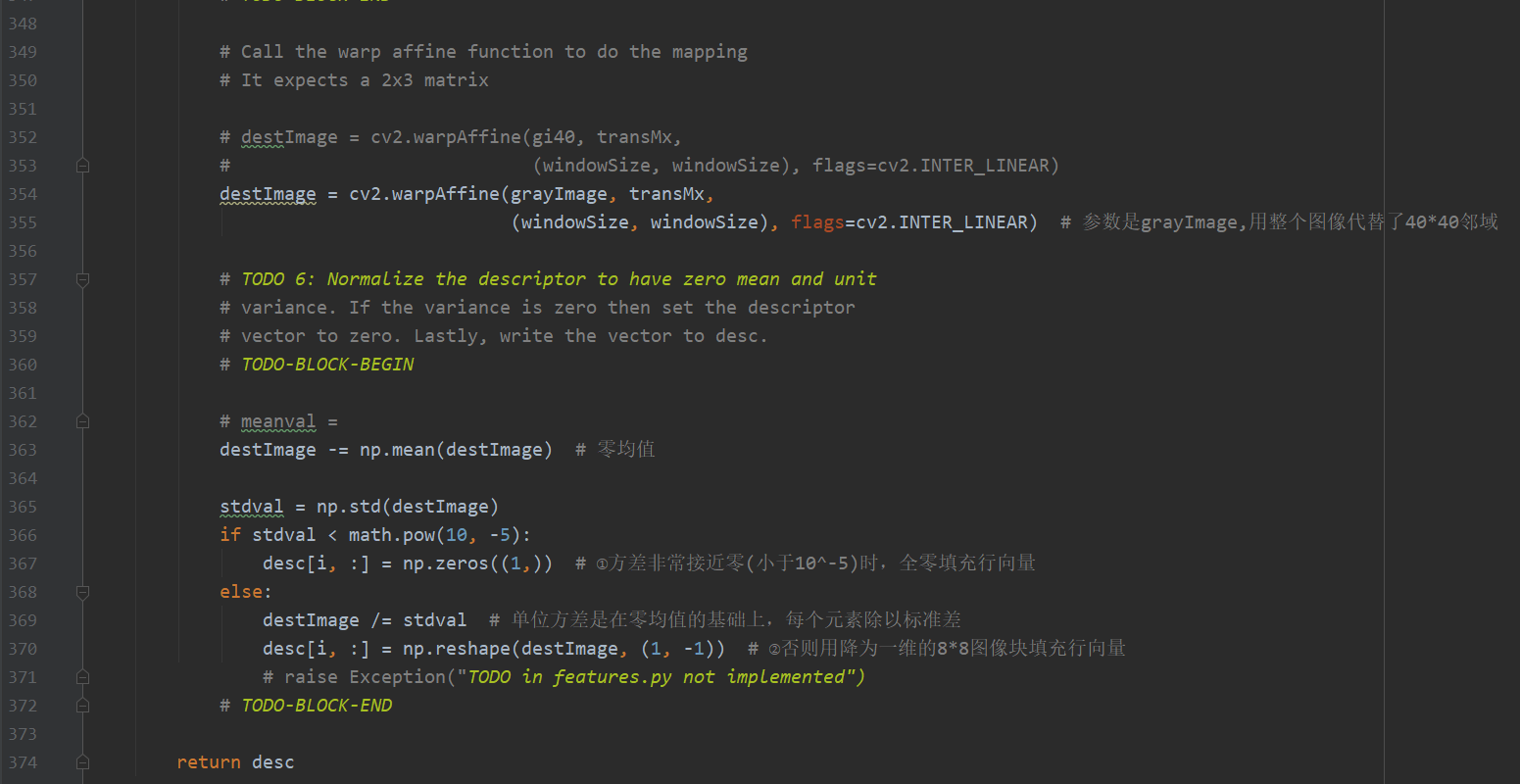


Simple特征描述：

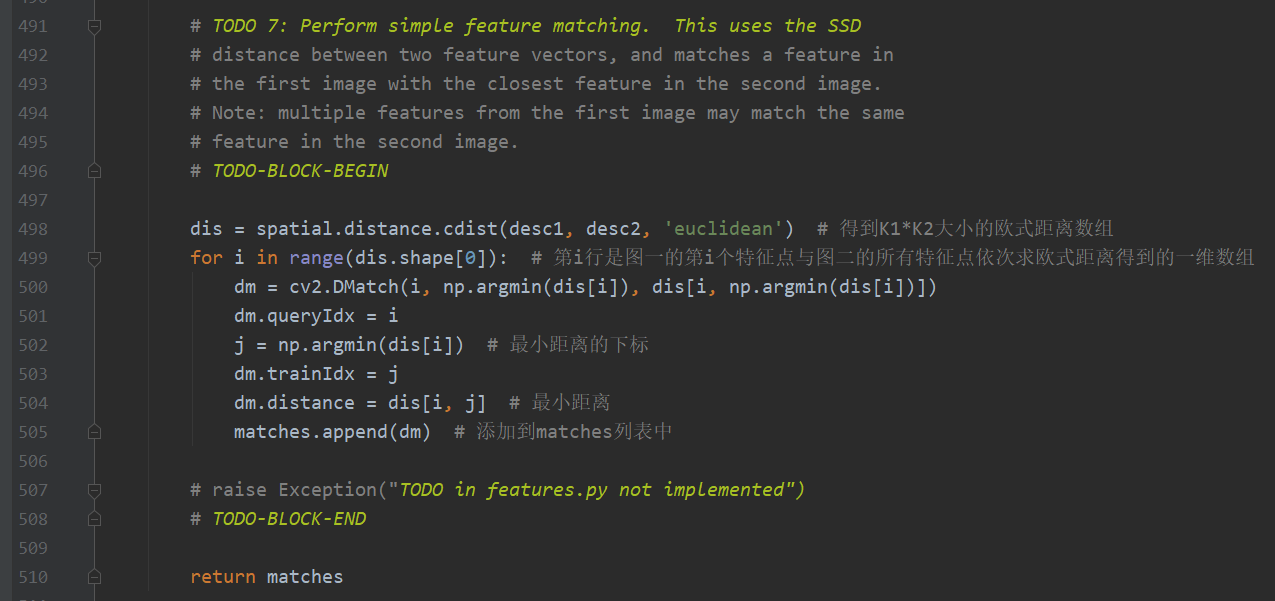


MOPS:

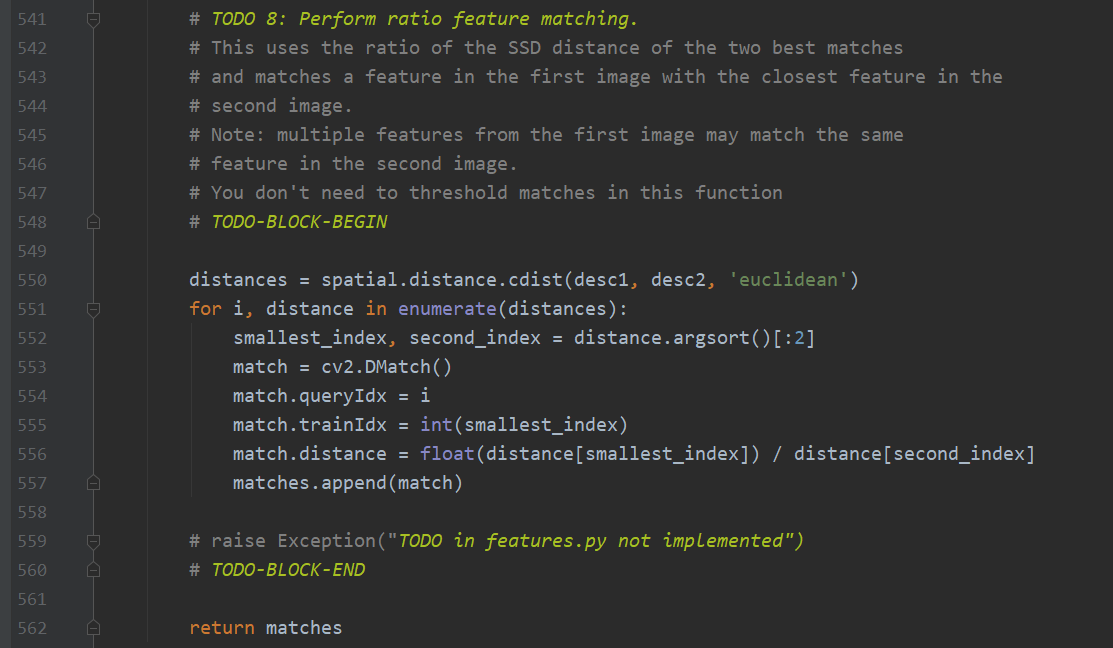




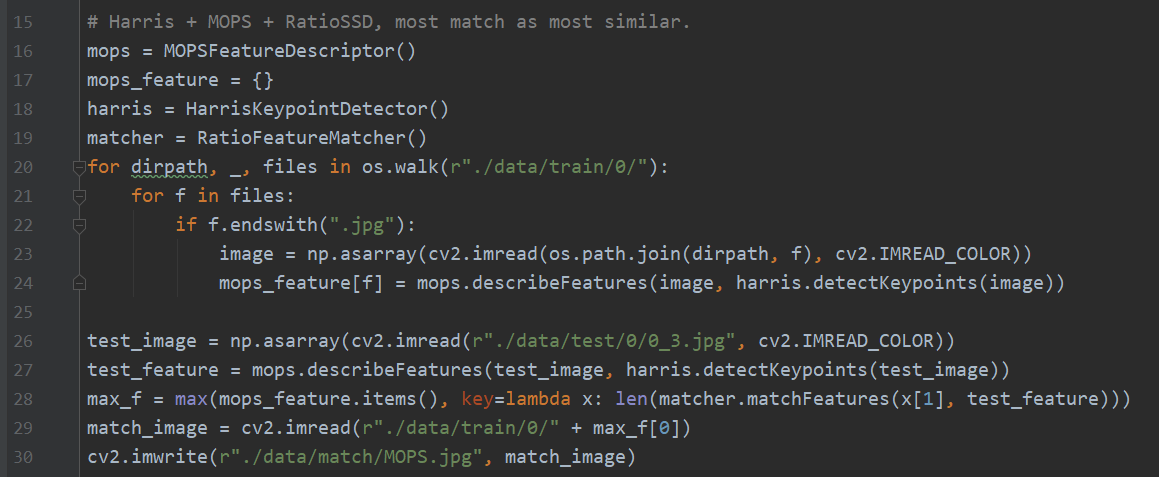
SSD:



RatioSSD:



Harris+MOPS+Ratio SSD：



测试数据：

输入：/data/test/0\_3.jpg/



输出：

MOPS结果：



SIFT结果：



# 第5次实验 KDTree

## 算法描述：

参考：

<https://leileiluoluo.com/posts/kdtree-algorithm-and-implementation.html>

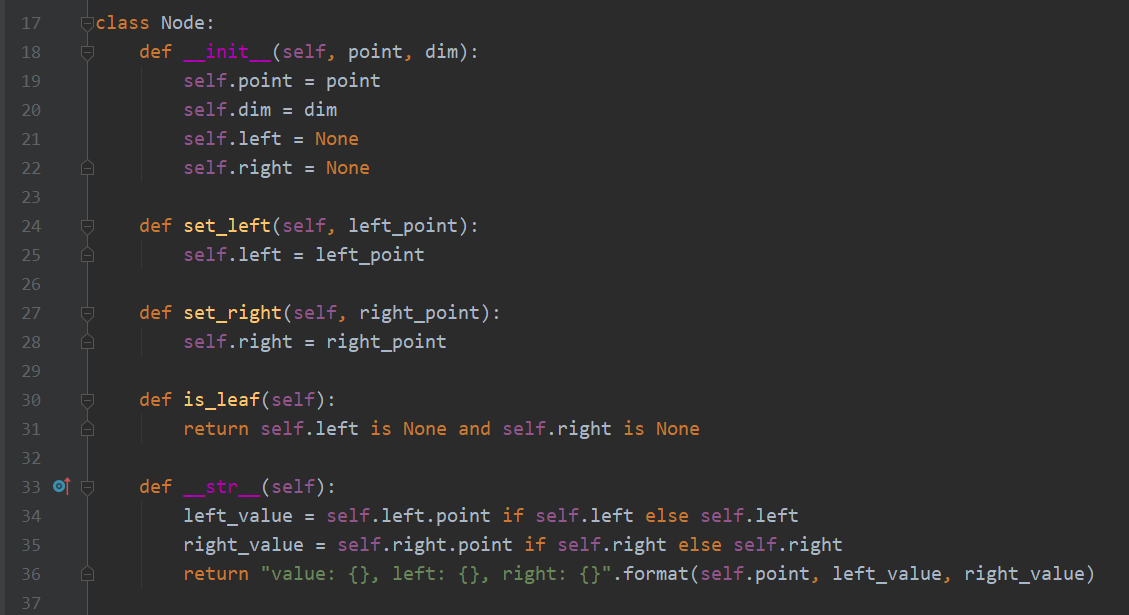
<https://www.jianshu.com/p/7d15944290fb>

在构造KD树，方法与普通二叉树类似，只需要注意每层二叉树维度的变化。选择树节点的数值时均进行一次在当前维度上的排序，并取中间值，从而提高查找效率。

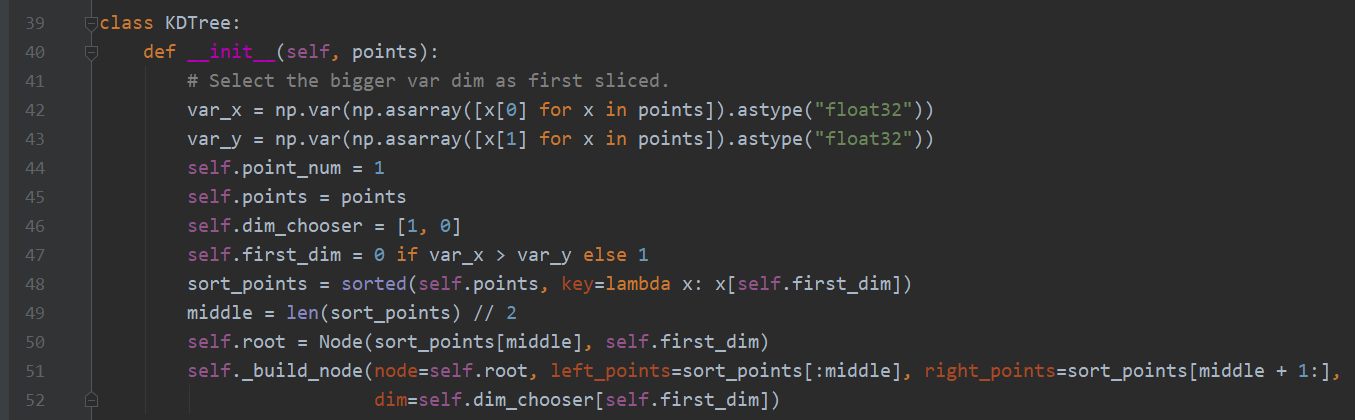
查找时，首先按照类似二叉查找树方式深度优先搜索，直到叶子节点。随后向上回溯，在回溯过程中判断当前节点是否与查找点有更小距离；同时判断最小距离是否大于查找点到当前节点划分超平面的平面（在二维下即为线）的垂直距离，如果大于，则代表超平面另一侧可能有节点存在更小距离，因此将另一侧节点加入，即加入当前节点在深度优先搜索中没有遍历过的孩子。循环直到搜索队列中不存在节点了。

## 核心源程序：

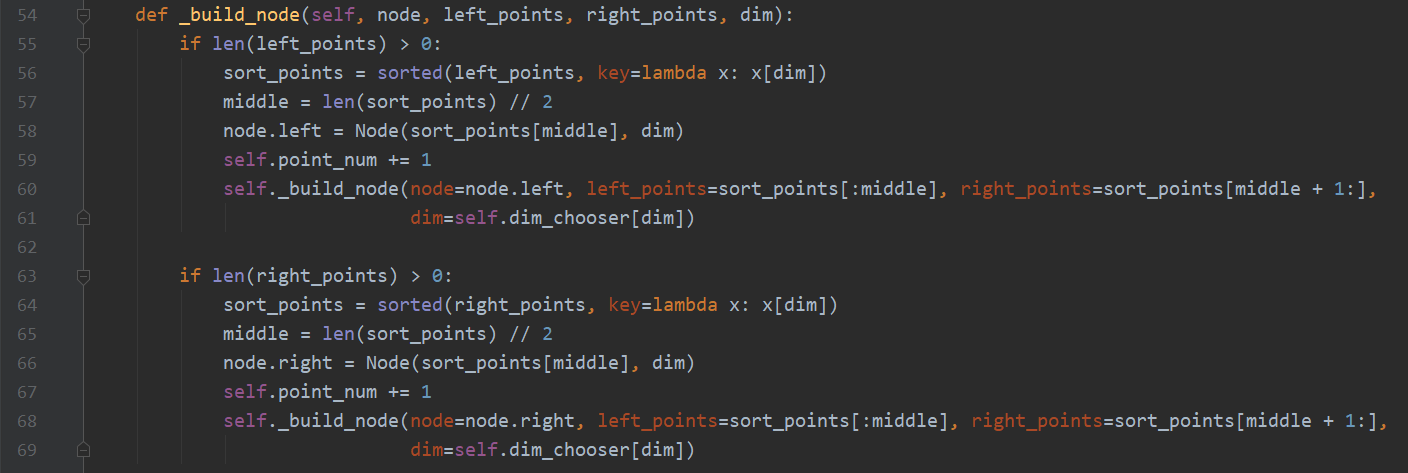
节点：类似二叉树



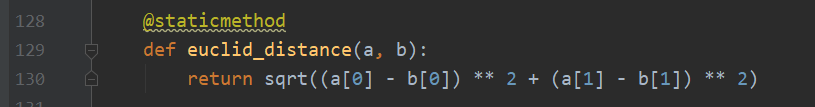
初始化：方差大的维度作为第一个划分维度，提高查找效率。利用dim\_chooser切换维度。



构造节点：采取递归。

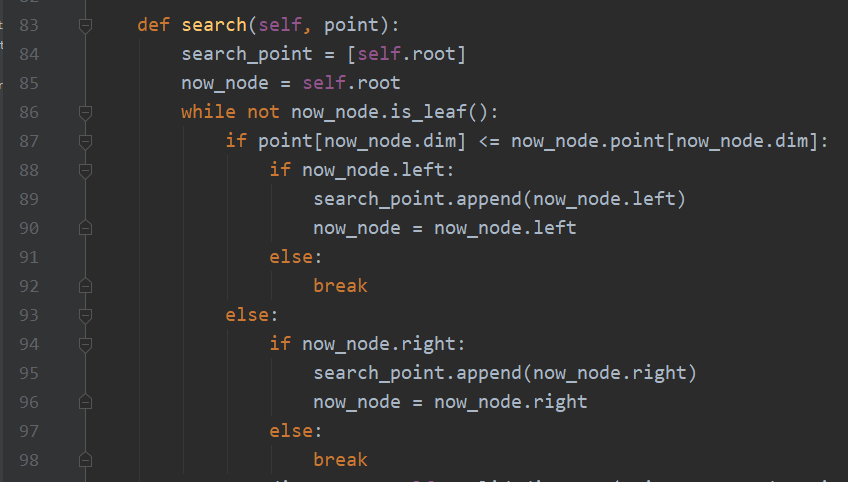


计算距离：欧氏距离

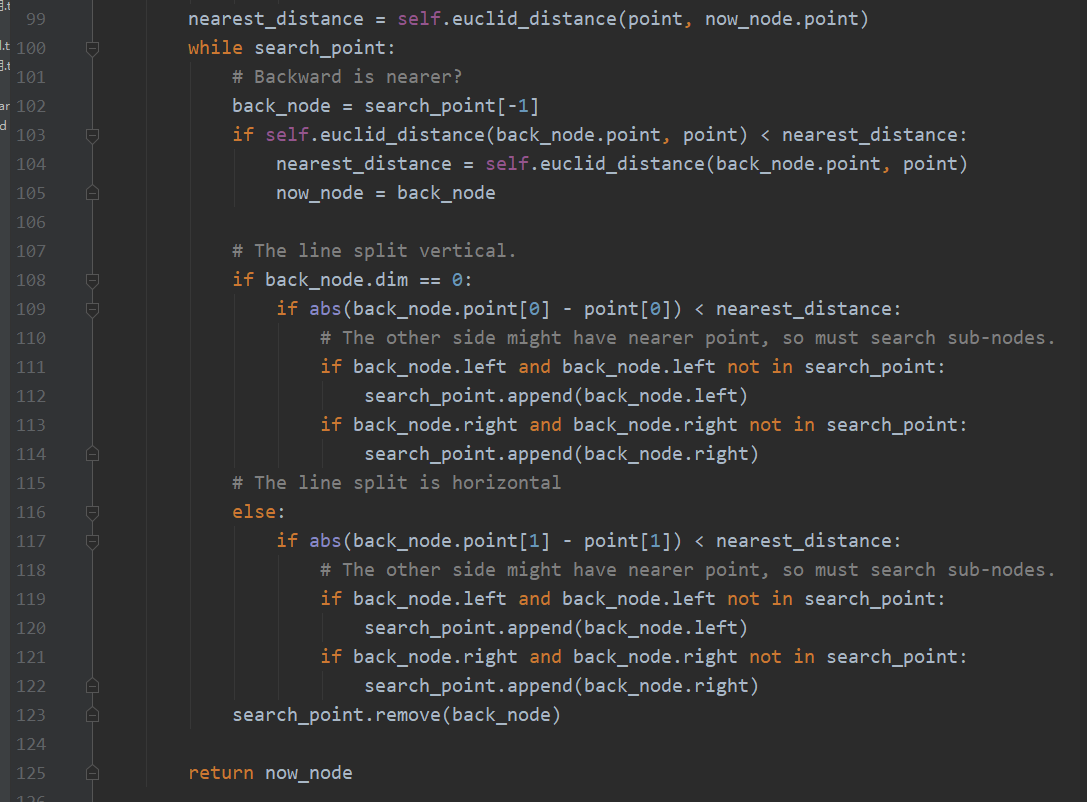


查找：

首先深度优先遍历。



随后反向遍历，并且添加需要查找的节点。



## 测试数据：

利用sklearn的KDTree作为验证

上为自己的KDT，下为sklearn

输入：

BJ数据集，

116.42, 39.97

输出：



输入：

CA数据集

-122.42, 37.89

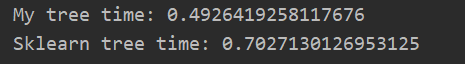
输出：



输入：

10000个CA数据集数据附近的点

输出：自己构造的树和sklearn树所用查找时间



# 个人感想

学习了很多优秀的算法和其设计思想，对于大规模数据管理的逻辑有了更深刻的认知；同时和计算机视觉，数据挖掘，机器学习等课程产生了交叉，相互加深了知识。

很多算法没有精力和能力去自己实现是一个遗憾，希望未来可以增强自己研究算法，读取论文，分析资料的能力。

课程建议：实验部分提供一些小型实验作为作业，例如数据科学包的使用，一些相对简单的算法，循序渐进提高学生的编程能力。